

**Lab Report**

**Course Name:** Artificial Intelligence

**Name Of Teacher:** Dr. Yanmei Zheng(郑艳梅)

**Student ID:** 10460348612

**Student's Name:** KAIMUZZAMAN(扎尔曼）

**Professional Class:** CST-Batch2018

**Session ( 2021-2022-1 )**

|  |
| --- |
| **Henan Polytechnic University**  **Score Standard of Lab Report** |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | IDX | Evaluation Index | Points | Evaluation grade and reference score | | | | | Points | | Excellent | Good | Medium | Qualified | Poor | | 1 | The experimental report is complete and substantial | 10 | 10 | 8 | 7 | 6 | 3 |  | | 2 | The experiments are written in a standard and neat way | 10 | 10 | 8 | 7 | 6 | 3 |  | | 3 | Detailed description of the experiment process, correct concept, accurate language expression, rigorous structure, clear and logical, no plagiarism. | 30 | 30 | 26 | 23 | 20 | 10 |  | | 4 | Analyze the problems in the process of the experiment in detail, thoroughly, profoundly, comprehensively and standardized. Have personal opinions and ideas, and can put forward relevant problems and give solutions. | 30 | 30 | 26 | 23 | 20 | 10 |  | | 5 | The experimental results, analysis and conclusion are correct | 20 | 20 | 17 | 15 | 13 | 6 |  | | Total Points: | | | | | | | |  |   **Signature (seal):**  **Date:** |
|  |

|  |
| --- |
| **Lab Report of Henan Polytechnic University** |
| **Date: 12.05** |
| **Lab Name: A\*** |
| **Lab Aim And Requirement:**  **1) Be familiar with and master the definition of heuristic search, evaluation function and algorithm process.**  **2) Use A\* algorithm to solve N-number problems, understand the solution process and search order.**  **3) Master related functions of numpy library.** |
| **Lab Steps:**  1. As Python IDE, I used Jupyter notebook. So I opened a new file.    2. Named the file as Astar-8-Puzzle and saved it as .py file    **3.** Wrote all the code. Below some picture of the code          1.For getting the location:  def get\_location(vec, num):  row\_num = vec.shape[0]  line\_num = vec.shape[1]  for i in range(row\_num):  for j in range(line\_num):  if num == vec[i][j]:  return i, j  2.Getting position of num matirix.  def get\_actions(vec):  row\_num = vec.shape[0]  line\_num = vec.shape[1]  (x, y) = get\_location(vec, 0)  action = [(0, 1), (0, -1), (1, 0), (-1, 0)]  if x == 0:  on the position  action.remove((-1, 0))  if y == 0:  action.remove((0, -1))  if x == row\_num - 1:  action.remove((1, 0))  if y == line\_num - 1:  action.remove((0, 1))  return list(action)  def result(vec, action):  (x, y) = get\_location(vec, 0)  (a, b) = action  n = vec[x+a][y+b]  s = copy.deepcopy(vec)  s[x+a][y+b] = 0  s[x][y] = n  return s   1. Manhattan distance of two matrices is calculated.   def get\_ManhattanDis(vec1, vec2):  row\_num = vec1.shape[0]  line\_num = vec1.shape[1]  dis = 0  for i in range(row\_num):  for j in range(line\_num):  if vec1[i][j] != vec2[i][j] and vec2[i][j] != 0:  k, m = get\_location(vec1, vec2[i][j])  d = abs(i - k) + abs(j - m)  dis += d  return dis   1. Expanding the node:   def expand(p, actions, step):  children = []  for action in actions:  child = {}  child['parent'] = p  child['vec'] = (result(p['vec'], action))  child['dis'] = get\_ManhattanDis(goal['vec'], child['vec'])  child['step'] = step + 1  child['dis'] = child['dis'] + child['step']  child['action'] = get\_actions(child['vec'])  children.append(child)  return children   1. Sorting and getting the input:   def node\_sort(nodelist):  return sorted(nodelist, key = itemgetter('dis'), reverse=True)  def get\_input(num):  A = []  for i in range(num):  temp = []  p = []  s = input()  temp = s.split(' ')  for t in temp:  t = int(t)  p.append(t)  A.append(p)  return A  def get\_parent(node):  q = {}  q = node['parent']  return q    def test():  openlist = []  close = []  print('Please enter the number of rows of the matrix')  num=3;  print("Please enter the initial matrix A")  A=np.mat('1 0 2;4 5 6;3 7 8')  print("Please enter the target matrix B")  B=np.mat('1 2 3;8 0 4;7 6 5')  print("Please enter the filename of the result")  resultfile = "a.txt"  goal['vec'] = np.array(B)  p = {}  p['vec'] = np.array(A)  p['dis'] = get\_ManhattanDis(goal['vec'], p['vec'])  p['step'] = 0  p['action'] = get\_actions(p['vec'])  p['parent'] = {}  if (p['vec'] == goal['vec']).all():  return  openlist.append(p)  while openlist:  children = []  node = openlist.pop()  close.append(node)  if (node['vec'] == goal['vec']).all():  h = open(resultfile,'w',encoding='utf-8',)  h.write('Size of the search tree:' + str(len(openlist)+len(close)) + '\n')  h.write('close：' + str(len(close)) + '\n')  h.write('openlist：' + str(len(openlist)) + '\n')  h.write('The path length:' + str(node['dis']) + '\n')  h.write('The path of the solution: ' + '\n')  i = 0  way = []  while close:  way.append(node['vec'])  node = get\_parent(node)  if(node['vec'] == p['vec']).all():  way.append(node['vec'])  break  while way:  i += 1  h.write(str(i) + '\n')  h.write(str(way.pop()) + '\n')  h.close()  f = open(resultfile,'r',encoding='utf-8',)  print(f.read())  return  children = expand(node, node['action'], node['step'])  for child in children:  f = False  flag = False  j = 0  for i in range(len(openlist)):  if (child['vec'] == openlist[i]['vec']).all():  j = i  flag = True  break  for i in range(len(close)):  if(child['vec'] == close[i]).all():  f = True  break  if f == False and flag == False :  openlist.append(child)  elif flag == True:  if child['dis'] < openlist[j]['dis']:  del openlist[j]  openlist.append(child)  openlist = node\_sort(openlist)  test() |
| **Lab Result:** |
| **Lab Analysis:**  Block Diagram of A\* algorithm:  IMG_256  Through this experiment, I learned the basic idea of the A-star algorithm, got a preliminary understanding of the AI algorithm. The application of the star algorithm has a deeper understanding; the comparison of different valuation functions, and the understanding of the calculation of different valuation functions. A\* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A\* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A\* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).  In A\* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number.  In code we used get\_location() function to get the position of num in the matrix then get\_action() function to find next position where current position can move. Later, get\_ManhattanDis() function is used to find the distance between two matrices. Inside expand() function, p is the current matrix, step indicates the number of step taken and actions is the list of extensible states for the current matrix. Node\_sort() is used to sort the nodes form large to small then the get\_input(), get\_parent() and test() function is used. |
| **Grade:**  **Date:** |

|  |
| --- |
| **Lab Report of Henan Polytechnic University** |
| **Date: 12.12** |
| **Lab Name:** **The genetic algorithm** |
| **Lab Aim And Requirement:**  **1) Be familiar with and master the principle, process and coding strategy of genetic**  **algorithm**  **2) Use genetic algorithm to solve function optimization problem**  **3) Understand the process of solving the TSP problem and test the impact of major**  **parameters on the results.** |
| **Lab Steps:**  **1.** Coding: through coding, the problem variable to be solved is expressed as genotype string structure data-chromosome;  **2.** Generate the initial population: N chromosomes are randomly generated after encoding to construct the initial population of the genetic algorithm, and then use the  Start the iterative search with the initial population as the starting point;  **3.** Calculate the fitness of the individual: use the fitness function to evaluate the pros and cons of the solution;  **4.** Selection: also known as duplication or reproduction, that is, to select a chromosome with strong vitality from the current population, so that it has the opportunity to retain its use.  To breed offspring;  **5**. Crossover; also known as recombination or pairing, that is, from individuals in the chromosomes used for reproduction, randomly cross and interchange the dyes in the individuals  Mutation: Mutation typically works by making very small changes at random to an individual’s genome.  **6.** Repeat : Now we have our next generation we can start again from step two until we reach a termination condition  **7.** Plotting the best path found  **Code:**  **#.**Importing the relevant package and declaration of variable:  from random import shuffle, random, randint  from math import sqrt, floor  import matplotlib.pyplot as plt  # constants  GENERATION\_COUNT = 1000  POPULATION\_COUNT = 500  SATURATION\_PERCENTAGE = 0.5  MUTATION\_PROBABILITY = 0.9  **#** Giving the city coordinates:  coordinates = [(20,20), (20,40), (20,160), (40,120),(60,20), (60,80), (60,200), (80,180), (100,40), (100,120), (100,160), (120,80), (140,140), (140,180), (160,20), (180,60), (180,100), (180,200), (200,40), (200,160)]  **#** creating a path object:  class Path:  def \_\_init\_\_(self, sequence):  self.sequence = sequence  self.distance = 0  self.fitness = 0  def \_\_repr\_\_(self):  return "{ " + f"Path: {self.sequence}, Fitness: {self.fitness}" + " }"  **#** initialization: Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.  def initialization(path, populationCount):  population = [path]  for i in range(populationCount - 1):  newPath = path.sequence[:]  while pathExists(newPath, population):  shuffle(newPath)  population.append(Path(newPath))  return population  **#** Returns true if the path exists and false otherwise  def pathExists(path, population):  for item in population:  if item.sequence == path:  return True  return False  **#** Evaluation: Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'  def calculateDistance(path):  total = 0  for i in range(len(path.sequence)):  if i == len(path.sequence) - 1:  distance = sqrt((coordinates[path.sequence[0]][0] - coordinates[path.sequence[i]][0])\*\*2 + (  coordinates[path.sequence[0]][1] - coordinates[path.sequence[i]][1])\*\*2)  total += distance  else:  distance = sqrt((coordinates[path.sequence[i+1]][0] - coordinates[path.sequence[i]][0])\*\*2 + (  coordinates[path.sequence[i+1]][1] - coordinates[path.sequence[i]][1])\*\*2)  total += distance  path.distance = total  return total  def calculateFitness(population):  sum = 0  for path in population:  distance = calculateDistance(path)  sum += 1/distance  path.fitness = 1/distance  for path in population:  path.fitness /= sum  return sorted(population, key=lambda x: x.fitness, reverse=True)  **#** Selection: We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.  def select(population):  randomNumber = random()  third = floor(0.3 \* len(population))  randomIndex = randint(0, third)  if randomNumber <= 0.7:  return population[randomIndex]  else:  return population[randint(third+1, len(population) - 1)]  **#** Crossover: During crossover we create new individuals by combining aspects of ourselected individuals. from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from We can think of this as mimicking how sex works in nature. The hope is that by combining certain traits each of its parents.  def crossOver(population):  father = select(population)  mother = select(population)  while(mother == father):  mother = select(population)  startIndex = randint(0,len(mother.sequence) - 2)  endIndex = randint(startIndex + 1, len(mother.sequence) - 1)  childSequence = [None] \* len(population[0].sequence)  for i in range(startIndex, endIndex + 1):  childSequence[i] = mother.sequence[i]  for i in range(len(childSequence)):  if childSequence[i] is None:  for j in range(0, len(childSequence)):  if father.sequence[j] not in childSequence:  childSequence[i] = father.sequence[j]  break  return Path(childSequence)  def crossOverTwoHalfandHalf(population):  father = select(population)  mother = select(population)  while(mother == father):  mother = select(population)  mid = len(mother.sequence) // 2  childSequence = [None] \* len(mother.sequence)  for i in range(mid):  childSequence[i] = mother.sequence[i]  for i in range(mid, len(father.sequence)):  for k in range(len(father.sequence)):  if father.sequence[k] not in childSequence:  childSequence[i] = father.sequence[k]  break  return Path(childSequence)  **#** Mutation :We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individual’s genome.  def mutation(path):  firstIndex = randint(0, len(path.sequence) - 1)  secondIndex = randint(0, len(path.sequence) - 1)  while secondIndex == firstIndex:  secondIndex = randint(0, len(path.sequence) - 1)  probability = random()  if probability < MUTATION\_PROBABILITY:  temp = path.sequence[firstIndex]  path.sequence[firstIndex] = path.sequence[secondIndex]  path.sequence[secondIndex] = temp  return path  def mutationTwoInsertion(path):  firstIndex = randint(0, len(path.sequence) - 1)  secondIndex = randint(0, len(path.sequence) - 1)  while secondIndex == firstIndex:  secondIndex = randint(0, len(path.sequence) - 1)  probability = random()  if probability < MUTATION\_PROBABILITY:  city = path.sequence[firstIndex]  path.sequence.remove(path.sequence[firstIndex])  path.sequence.insert(secondIndex, city)  return path  **#** Repeat : Now we have our next generation we can start again from step two until we reach a termination condition  def geneticAlgorithm(path, populationCount, generationCount):  path = Path(path)  population = initialization(path, populationCount)  population = calculateFitness(population)  best = population[0]  print(f"Generation 1: Fitness: {best.fitness}, Distance: {round(best.distance, 2)}")  saturation = 0  for i in range(2, generationCount + 1):  print(f"Generation {i}: Fitness: {best.fitness}, Distance: {round(best.distance, 2)}")  newGeneration = []  for \_ in range(populationCount):  child = crossOver(population)  # child = crossOverTwoHalfandHalf(population)  newGeneration.append(mutation(child))  # newGeneration.append(mutationTwoInsertion(child))  population = calculateFitness(newGeneration)  if population[0].fitness > best.fitness:  best = population[0]  saturation = 0  else:  saturation += 1  if saturation > (SATURATION\_PERCENTAGE \* GENERATION\_COUNT):  break  return best  # Plotting the best path found  def plotData(path):  x = []  y = []  for i in range(len(path.sequence)):  x.append(coordinates[path.sequence[i]][0])  y.append(coordinates[path.sequence[i]][1])  x.append(coordinates[path.sequence[0]][0])  y.append(coordinates[path.sequence[0]][1])  plt.xlabel("x")  plt.ylabel("y")  plt.title(f"Traveling Sales Person\nSequence:\n{path.sequence}")  plt.plot(x, y, "bo-")  plt.show()  **#** program entry point  if \_\_name\_\_ == "\_\_main\_\_":  cities = list(range(20))  best = geneticAlgorithm(cities, POPULATION\_COUNT, GENERATION\_COUNT)  plotData(best) |
| **Lab Result:** |
| **Lab Analysis:**  **Flow Chart of Genetic Algorithm:**  IMG_256  Basically realized the genetic algorithm process, and realized the visualization with matplotlib, and carried out the parameters in the experiment.  The gradient is increased and the graph is compared. Different parameters have different effects on the genetic algorithm. Therefore, parameter tuning is very important in the AI algorithm, but there are also shortcomings. Sometimes the final result cannot be found, but it is just close to the final result.  In the experiment, a comparative analysis of the single-variable principle was carried out on different parameters. I found that the number of iterations has an effect on the accuracy of the algorithm.  The accuracy has a great influence. When the number of iterations is too small, the accuracy of the result will be lower. Therefore, for different problem scales, it is necessary to set  Reasonable parameters.  Through this experiment, I found that if you want to get the best plan of the travel route, you must have parameters suitable for the algorithm.  It also illustrates the importance of tuning. By writing this algorithm, I also improved my programming ability and my perspective on the problem.  Methods used to solve different problems. The disadvantage is that the efficiency of the algorithm is relatively low, and the result of the operation is only close to the final result.  As a result, the efficiency of the algorithm needs to be improved. |
| **Grade:**  **Date:** |

|  |
| --- |
| **Lab Report of Henan Polytechnic University** |
| **Date: 12.19** |
| **Lab Name:** **Animal Recognition** |
| **Lab Aim And Requirement:**  **1) Be familiar with the representation of knowledge**  **2) Master the operating mechanism of the production system**  **3) Basic methods of production system reasoning.**  Using the learned knowledge, design and program a small animal recognition system, so that  this production system can identify the tiger, leopard, zebra, giraffe, ostrich, penguin, and  albatross.  Rule base：  R1: IF The animal has hair THEN The animal is a mammal  R2: IF The animal has milk THEN The animal is a mammal  R3: IF The animal has feathers THEN The animal is a bird  R4: IF The animal can fly AND The animal lays eggs, THEN The animal is a bird  R5: IF The animal eats meat THEN The animal is a carnivore  R6: IF The animal has canine teeth AND The animal has claws AND The animal gazes ahead  THEN The animal is a carnivore  R7: IF The animal is a mammal AND The animal has hooves THEN The animal is an ungulate  R8: IF The animal is a mammal AND The animal is a ruminant THEN The animal is an ungulate  R9: IF The animal is a mammal AND The animal is a predator AND The animal is tawny AND There are dark spots on its body THEN The animal is the leopard  R10: IF The animal is a mammal AND The animal is a predator AND The animal is tawny AND There are black stripes on its body THEN The animal is a tiger  R11: IF The animal is an ungulate AND The animal has a long neck AND The animal has long legs AND There are dark spots on its body THEN The animal is a giraffe  R 12: IF The animal is an ungulate AND There are black stripes on its body THEN The animal is a zebra  R13: IF The animal is a bird AND The animal has a long neck AND The animal has long legs AND The animal can't fly AND The animal is black and white THEN The animal is an ostrich  R14: IF The animal is a bird AND The animal can swim AND The animal can't fly AND The animal is black and white THEN The animal is a penguinR15: IF The animal is a bird AND The animal is good at flying THEN The animal is an albatross |
| **Lab Steps:**   1. Creating an indirect rule base where the individual animal will be added   def addup\_indirect\_ruleslibrary(list, key1, key2, value1, value2):  while (1):  str1 = input("Please enter animal properties (space separated, end with 0): ")  if (str1 == '0'): break  a = str1.split()  key1.append(a)  str2 = input("Please enter the result：")  value1.append(str2)  len1 = len(a)  for i in range(0, len1):  if i not in list:  list.append(a[i])  return 1   1. Create direct rule base and adding individual attributes of the animal to the rule base   def addup\_direct\_ruleslibrary(list, key1, key2, value1, value2):  while (1):  str1 = input("Please enter animal properties (space separated, end with 0): ")  if (str1 == '0'): break  a = str1.split()  key2.append(a)  str2 = input("Please enter the result：")  value2.append(str2)  len2 = len(a)  for i in range(0, len2):  if i not in list:  list.append(a[i])  return 1   1. Animal identification and initialize the comprehensive database   def recognize(list, key1, key2, value1, value2):  map = { }  len1 = len(list)  for i in range(0, len1): map[list[i]] = 0  str = input("Please enter animal properties :(space separated)")  list1 = str.split()  len1 = len(list1)  for i in range(0, len1): map[list1[i]] = 1  len1 = len(key1)  for i in range(0, len1):  list2 = key1[i]  len2 = len(list2)  flag = 1  for j in range(0, len2):  if(map[list2[j]] == 0):  flag = 0  break  if(flag):  map[value1[i]] = 1  len1 = len(key2)  for i in range(0, len1):  list2 = key2[i]  len2 = len(list2)  flag = 1  for j in range(0, len2):  if (map[list2[j]] == 0):  flag = 0  break  if (flag):  return value2[i]  return "Failed identification ! "   1. Solve:   def solve(list, key1, key2, value1, value2):  while(1):  print("\n1. Add the direct rule library."  "2. Add the indirect rule library."  "3. Do animal identification."  "4. Exit the program!\n")  n = int(input("Please select ："))  if (n == 1):  addup\_direct\_ruleslibrary(list, key1, key2, value1, value2)  elif(n == 2):  addup\_indirect\_ruleslibrary(list, key1, key2, value1, value2)  elif(n == 3):  str = recognize(list, key1, key2, value1, value2)  print("The animal is：",str)  elif(n == 4):  print("\nSuccessful exit procedure！\n")  break  else:  print("\nPlease re-enter！\n")  return 1   1. Main function to store in rule, indirect, direct base antecedents and also initialize   def main():  list = [ ]  key1 = [ ]  key2 = [ ]  value1 = [ ]  value2 = [ ]  init(list, key1, key2, value1, value2)  solve(list, key1, key2, value1, value2) |
| **Lab Result:** |
| **Lab Analysis:**  This lab was really fun and I enjoyed this lab very much because this lab gave me feeling of creating machine which can give a solution with the data it was fetched. We actually did production system experiment which means this system is based on a set of rules about behavior. These rules are a basic representation found helpful in expert systems, automated planning, and action selection. I also learned about the two kind of database, first one is indirect and second one is direct database. A two‐dimensional list and an one‐dimensional list are used to store the key and value. The second part is to add the database and recognize the animal, added when being queried, can effectively expand the database, make the database more flexible and perfect. The complexity of the query is O(1). Creating the base is pretty tough and the data entry need to be specific and error less because if one attribute is wrong then the machine will not be able to identify the animal clearly that’s why I needed to be more cautious while adding the information about the animals. Although this production system not so highly accurate compared with the other high performing machine because the information about a animal I used is really little knowledge about that animal. So if I want to use this system for the real life testing purpose I need to add more properties to the database for every animal. Although, while running the code the machine was able to identify the bird albatross. For future using, this machine need more improvement. |
| **Grade:**  **Date:** |

|  |
| --- |
| **Lab Report of Henan Polytechnic University** |
| **Date: 12.26** |
| **Lab Name:** **Savage and Missionary** |
| **Lab Aim And Requirement:**  **1. Understand and master the depth-first search algorithm**  **2. Understand the idea of recursion** |
| **Lab Steps:**  **1.**Problem analysis  Suppose the number N of missionaries and wild men is 3, and the maximum passenger capacity K of the ship at one time is 2 for analysis.  Initial state: 3 savages and 3 missionaries on the left bank of the river; 0 savages and 0 missionaries on the right bank of the river; the boat stops at  On the left bank, there are 0 people on board.  Target status: 0 savages and 0 missionaries on the left bank of the river; 3 savages and 3 missionaries on the right bank of the river; the boat stops at  On the right bank, there are 0 people on board.  The whole problem is abstracted into how to go from the initial state to a series of intermediate states to reach the goal state, state change  It was triggered by boating across the river.  According to the requirements, a total of 5 possible river crossing plans are drawn as follows:  (1) Crossing 2 Missionaries  (2) Cross 2 Savage  (3) Crossing 1 Savage 1 Missionary  (4) Cross 1 missionary  (5) Crossing 1 Savage  This program uses classes to define state nodes, uses collections to store state nodes, and uses the idea of recursion (depth first query) To find the target state.  **2.** Constructing the state space  The state set is (m, c, b) triples, c represents the number of wild people on the left bank, m represents the number of missionaries on the left bank, and m, c take values from 0 to 3. b is 0  Means the ship is on the left, b is 1 means the ship is on the right  The action set is one missionary from left to right, two missionaries from left to right, one savages from left to right, and two savages from left  To the right, a savage and a missionary from left to right; from right to left, there are similarly 5 actions, a total of 10 actions, so  You can draw a state transition diagram. The initial state is (3, 3, 1), and the destination state is (0, 0, 0).  **3**.Def safe(s): #weather the state is safe  def safe(s):  if s.m > M or s.m < 0 or s.c > C or s.c < 0 or (s.m != 0 and s.m < s.c) or (s.m != M and M - s.m < C - s.c):  return False  else:  return True  **4.**def back(new, s): # Determine weather current state is consistent with parent state  if s.father is None:  return False  #Determines whether the current state is consistent with the ancestor state  c=b=s.father  while(1):  a,c=equal(new, b)  if a:  return True  b=c.father  if b is None:  return False  **5**.# Recursive print path  def printPath(f):  if f is None:  return  printPath(f.father)  print(f.node ) |
| **Lab Result:** |
| **Lab Analysis:**  In this lab we learned the use of depth-first search algorithm and also recursion . Initial and target states: (3, 3, 1) and (0, 0, 0), the same as the problem of eight digital games, a production system description is established. After that, the state space can be searched through the control strategy, and a ferry operation sequence can be obtained to achieve the target state.  When discussing the use of production systems to solve problems, it is sometimes helpful to introduce the concept of state space diagrams. The state space diagram is a directed graph whose nodes can represent various states of the problem, and the arcs between nodes represent some operations (production rules), they can lead one state to another. The state space diagram established in this way describes all possible problems. The state and the relationship between state and operation, so the solution path and nature of the problem can be seen more intuitively.  To establish the state space,we set the state variable like, in left bank missionaries m={0,1,2,3} and right side, 3-m. Cannibals in left c={0,1,2,3}, and right 3-c. Boat b={0,1} in left and in right 1-b. So, M<=N, C<=N, boat =K, so that M>=C and M+C<=K. We followed a set of operation and rule set. Some state were illegal because the number of cannibals can not be greater than missionaries and also four states were impossible because ships can not dock on uninhabited shores. After running the code we can see there are total four solution which means in four way the missionaries and cannibal can cross the river without violating the requirements. |
| **Grade:**  **Date:** |